Introduction

async-std

This book serves as high-level documentation for `async-std` and a way of learning async programming in Rust through it. As such, it focuses on the `async-std` API and the task model it gives you.

Please note that the Rust project provides its own book on asynchronous programming, called "Asynchronous Programming in Rust", which we highly recommend reading along with this book, as it provides a different, wider view on the topic.

Welcome to async-std

`async-std`, along with its supporting libraries, is a library making your life in async programming easier. It provides fundamental implementations for downstream libraries and applications alike. The name reflects the approach of this library: it is as closely modeled to the Rust main standard library as possible, replacing all components by async counterparts.

`async-std` provides an interface to all important primitives: filesystem operations, network operations and concurrency basics like timers. It also exposes a `task` in a model similar to the `thread` module found in the Rust standard lib. But it does not only include I/O primitives, but also `async/await` compatible versions of primitives like `Mutex`.

`std::future` and `futures-rs`

Rust has two kinds of types commonly referred to as `Future`:

- the first is `std::future::Future` from Rust's standard library.
- the second is `futures::future::Future` from the futures-rs crate.

The future defined in the futures-rs crate was the original implementation of the type. To enable the `async/await` syntax, the core Future trait was moved into Rust's standard library and became `std::future::Future`. In some sense, the `std::future::Future` can be seen as a minimal subset of `futures::future::Future`.

It is critical to understand the difference between `std::future::Future` and `futures::future::Future`, and the approach that `async-std` takes towards them. In itself, `std::future::Future` is not something you want to interact with as a user—except by calling `.await` on it. The inner workings of `std::future::Future` are mostly of interest to people implementing `Future`. Make no mistake—this is very useful! Most of the functionality that used to be defined on `Future` itself has been moved to an extension trait called `FutureExt`. From this information, you might be able to infer that the `futures` library serves as an extension to the core Rust async features.

In the same tradition as `futures`, `async-std` re-exports the core `std::future::Future` type. You can actively opt into the extensions provided by the `futures` crate by adding it to your `Cargo.toml` and importing `FutureExt`.

Interfaces and Stability

`async-std` aims to be a stable and reliable library, at the level of the Rust standard library. This also means that we don't rely on the `futures` library for our interface. Yet, we appreciate that many users have come to like the conveniences that `futures-rs` brings. For that reason, `async-std` implements all `futures` traits for its types.

Luckily, the approach from above gives you full flexibility. If you care about stability a lot, you can just use `async-std` as is. If you prefer the `futures` library interfaces, you link those in. Both uses are first class.

`async_std::future`

There's some support functions that we see as important for working with futures of any kind. These can be found in the `async_std::future` module and are covered by our stability guarantees.

Streams and Read/Write/Seek/BufRead traits

Due to limitations of the Rust compiler, those are currently implemented in `async_std`,

but cannot be implemented by users themselves.

Stability and SemVer

`async-std` follows https://semver.org/.

In short: we are versioning our software as `MAJOR.MINOR.PATCH`. We increase the:

- MAJOR version when there are incompatible API changes,
- MINOR version when we introduce functionality in a backwards-compatible manner
- PATCH version when we make backwards-compatible bug fixes

We will provide migration documentation between major versions.

Future expectations

`async-std` uses its own implementations of the following concepts:

- `Read`
- `Write`
- `Seek`
- `BufRead`
- `Stream`

For integration with the ecosystem, all types implementing these traits also have an implementation of the corresponding interfaces in the `futures-rs` library. Please note that our SemVer guarantees don't extend to usage of those interfaces. We expect those to be conservatively updated and in lockstep.

Minimum version policy

The current tentative policy is that the minimum Rust version required to use this crate can be increased in minor version updates. For example, if `async-std` 1.0 requires Rust 1.37.0, then `async-std` 1.0.z for all values of z will also require Rust 1.37.0 or newer. However, `async-std` 1.y for y > 0 may require a newer minimum version of Rust.

In general, this crate will be conservative with respect to the minimum supported version of Rust. With `async/await` being a new feature though, we will track changes in a measured pace initially.

Security fixes

Security fixes will be applied to *all* minor branches of this library in all *supported* major revisions. This policy might change in the future, in which case we give a notice at least *3 months* ahead.

Credits

This policy is based on BurntSushi's regex crate.

Async concepts using async-std

Rust Futures have the reputation of being hard. We don't think this is the case. They are, in our opinion, one of the easiest concurrency concepts around and have an intuitive explanation.

However, there are good reasons for that perception. Futures have three concepts at their base that seem to be a constant source of confusion: deferred computation, asynchronicity and independence of execution strategy.

These concepts are not hard, but something many people are not used to. This base confusion is amplified by many implementations oriented on details. Most explanations of these implementations also target advanced users, and can be hard for beginners. We try to provide both easy-to-understand primitives and approachable overviews of the concepts.

Futures are a concept that abstracts over how code is run. By themselves, they do nothing. This is a weird concept in an imperative language, where usually one thing happens after the other - right now.

So how do Futures run? You decide! Futures do nothing without the piece of code *executing* them. This part is called an *executor*. An *executor* decides *when* and *how* to execute your futures. The `async-std::task` module provides you with an interface to such an executor.

Let's start with a little bit of motivation, though.

Futures

A notable point about Rust is *fearless concurrency*. That is the notion that you should be

empowered to do concurrent things, without giving up safety. Also, Rust being a low-level language, it's about fearless concurrency *without picking a specific implementation strategy*. This means we *must* abstract over the strategy, to allow choice *later*, if we want to have any way to share code between users of different strategies.

Futures abstract over *computation*. They describe the "what", independent of the "where" and the "when". For that, they aim to break code into small, composable actions that can then be executed by a part of our system. Let's take a tour through what it means to compute things to find where we can abstract.

Send and Sync

Luckily, concurrent Rust already has two well-known and effective concepts abstracting over sharing between concurrent parts of a program: `Send` and `Sync`. Notably, both the `Send` and `Sync` traits abstract over *strategies* of concurrent work, compose neatly, and don't prescribe an implementation.

As a quick summary:

- `Send` abstracts over *passing data* in a computation to another concurrent computation (let's call it the receiver), losing access to it on the sender side. In many programming languages, this strategy is commonly implemented, but missing support from the language side, and expects you to enforce the "losing access" behaviour yourself. This is a regular source of bugs: senders keeping handles to sent things around and maybe even working with them after sending. Rust mitigates this problem by making this behaviour known. Types can be `Send` or not (by implementing the appropriate marker trait), allowing or disallowing sending them around, and the ownership and borrowing rules prevent subsequent access.

- `Sync` is about *sharing data* between two concurrent parts of a program. This is another common pattern: as writing to a memory location or reading while another party is writing is inherently unsafe, this access needs to be moderated through synchronisation.[1] There are many common ways for two parties to agree on not using the same part in memory at the same time, for example mutexes and spinlocks. Again, Rust gives you the option of (safely!) not caring. Rust gives you the ability to express that something *needs* synchronisation while not being specific about the *how*.

Note how we avoided any word like *"thread"*, but instead opted for "computation". The full power of `Send` and `Sync` is that they relieve you of the burden of knowing *what* shares. At the point of implementation, you only need to know which method of sharing is appropriate for the type at hand. This keeps reasoning local and is not influenced by

whatever implementation the user of that type later uses.

`Send` and `Sync` can be composed in interesting fashions, but that's beyond the scope here. You can find examples in the Rust Book.

To sum up: Rust gives us the ability to safely abstract over important properties of concurrent programs, their data sharing. It does so in a very lightweight fashion; the language itself only knows about the two markers `Send` and `Sync` and helps us a little by deriving them itself, when possible. The rest is a library concern.

An easy view of computation

While computation is a subject to write a whole book about, a very simplified view suffices for us: A sequence of composable operations which can branch based on a decision, run to succession and yield a result or yield an error

Deferring computation

As mentioned above, `Send` and `Sync` are about data. But programs are not only about data, they also talk about *computing* the data. And that's what `Futures` do. We are going to have a close look at how that works in the next chapter. Let's look at what Futures allow us to express, in English. Futures go from this plan:

- Do X
- If X succeeded, do Y

towards:

- Start doing X
- Once X succeeds, start doing Y

Remember the talk about "deferred computation" in the intro? That's all it is. Instead of telling the computer what to execute and decide upon *now*, you tell it what to start doing and how to react on potential events in the... well... `Future`.

Orienting towards the beginning

Let's have a look at a simple function, specifically the return value:

```rust
fn read_file(path: &str) -> io::Result<String> {
    let mut file = File::open(path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}
```

You can call that at any time, so you are in full control on when you call it. But here's the problem: the moment you call it, you transfer control to the called function until it returns a value - eventually. Note that this return value talks about the past. The past has a drawback: all decisions have been made. It has an advantage: the outcome is visible. We can unwrap the results of the program's past computation, and then decide what to do with it.

But we wanted to abstract over *computation* and let someone else choose how to run it. That's fundamentally incompatible with looking at the results of previous computation all the time. So, let's find a type that *describes* a computation without running it. Let's look at the function again:

```rust
fn read_file(path: &str) -> io::Result<String> {
    let mut file = File::open(path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}
```

Speaking in terms of time, we can only take action *before* calling the function or *after* the function returned. This is not desirable, as it takes from us the ability to do something *while* it runs. When working with parallel code, this would take from us the ability to start a parallel task while the first runs (because we gave away control).

This is the moment where we could reach for threads. But threads are a very specific concurrency primitive and we said that we are searching for an abstraction.

What we are searching for is something that represents ongoing work towards a result in the future. Whenever we say "something" in Rust, we almost always mean a trait. Let's start with an incomplete definition of the `Future` trait:

```rust
trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>;
}
```

Looking at it closely, we see the following:

- It is generic over the `Output`.
- It provides a function called `poll`, which allows us to check on the state of the

current computation.
- (Ignore `Pin` and `Context` for now, you don't need them for high-level understanding.)

Every call to `poll()` can result in one of these two cases:

1. The computation is done, `poll` will return `Poll::Ready`
2. The computation has not finished executing, it will return `Poll::Pending`

This allows us to externally check if a `Future` still has unfinished work, or is finally done and can give us the value. The most simple (but not efficient) way would be to just constantly poll futures in a loop. There are optimisations possible, and this is what a good runtime does for you. Note that calling `poll` again after case 1 happened may result in confusing behaviour. See the futures-docs for details.

Async

While the `Future` trait has existed in Rust for a while, it was inconvenient to build and describe them. For this, Rust now has a special syntax: `async`. The example from above, implemented with `async-std`, would look like this:

```
async fn read_file(path: &str) -> io::Result<String> {
    let mut file = File::open(path).await?;
    let mut contents = String::new();
    file.read_to_string(&mut contents).await?;
    Ok(contents)
}
```

Amazingly little difference, right? All we did is label the function `async` and insert 2 special commands: `.await`.

This `async` function sets up a deferred computation. When this function is called, it will produce a `Future<Output = io::Result<String>>` instead of immediately returning a `io::Result<String>`. (Or, more precisely, generate a type for you that implements `Future<Output = io::Result<String>>`.)

What does `.await` do?

The `.await` postfix does exactly what it says on the tin: the moment you use it, the code will wait until the requested action (e.g. opening a file or reading all data in it) is finished. The `.await?` is not special, it's just the application of the `?` operator to the result of

`.await`. So, what is gained over the initial code example? We're getting futures and then immediately waiting for them?

The `.await` points act as a marker. Here, the code will wait for a `Future` to produce its value. How will a future finish? You don't need to care! The marker allows the component (usually called the "runtime") in charge of *executing* this piece of code to take care of all the other things it has to do while the computation finishes. It will come back to this point when the operation you are doing in the background is done. This is why this style of programming is also called *evented programming*. We are waiting for *things to happen* (e.g. a file to be opened) and then react (by starting to read).

When executing 2 or more of these functions at the same time, our runtime system is then able to fill the wait time with handling *all the other events* currently going on.

Conclusion

Working from values, we searched for something that expresses *working towards a value available later*. From there, we talked about the concept of polling.

A `Future` is any data type that does not represent a value, but the ability to *produce a value at some point in the future*. Implementations of this are very varied and detailed depending on use-case, but the interface is simple.

Next, we will introduce you to `tasks`, which we will use to actually *run* Futures.

[1] Two parties reading while it is guaranteed that no one is writing is always safe.

Tasks

Now that we know what Futures are, we want to run them!

In `async-std`, the `task` module is responsible for this. The simplest way is using the `block_on` function:

```
use async_std::{fs::File, io, prelude::*, task};

async fn read_file(path: &str) -> io::Result<String> {
    let mut file = File::open(path).await?;
    let mut contents = String::new();
    file.read_to_string(&mut contents).await?;
    Ok(contents)
}
```

```
fn main() {
    let reader_task = task::spawn(async {
        let result = read_file("data.csv").await;
        match result {
            Ok(s) => println!("{}", s),
            Err(e) => println!("Error reading file: {:?}", e)
        }
    });
    println!("Started task!");
    task::block_on(reader_task);
    println!("Stopped task!");
}
```

This asks the runtime baked into `async_std` to execute the code that reads a file. Let's go one by one, though, inside to outside.

```
async {
    let result = read_file("data.csv").await;
    match result {
        Ok(s) => println!("{}", s),
        Err(e) => println!("Error reading file: {:?}", e)
    }
};
```

This is an `async` *block*. Async blocks are necessary to call `async` functions, and will instruct the compiler to include all the relevant instructions to do so. In Rust, all blocks return a value and `async` blocks happen to return a value of the kind `Future`.

But let's get to the interesting part:

```
task::spawn(async { });
```

`spawn` takes a `Future` and starts running it on a `Task`. It returns a `JoinHandle`. Futures in Rust are sometimes called *cold* Futures. You need something that starts running them. To run a Future, there may be some additional bookkeeping required, e.g. whether it's running or finished, where it is being placed in memory and what the current state is. This bookkeeping part is abstracted away in a `Task`.

A `Task` is similar to a `Thread`, with some minor differences: it will be scheduled by the program instead of the operating system kernel, and if it encounters a point where it needs to wait, the program itself is responsible for waking it up again. We'll talk a little bit about that later. An `async_std` task can also have a name and an ID, just like a thread.

For now, it is enough to know that once you have `spawn`ed a task, it will continue running in the background. The `JoinHandle` is itself a future that will finish once the

`Task` has run to conclusion. Much like with `threads` and the `join` function, we can now call `block_on` on the handle to *block* the program (or the calling thread, to be specific) and wait for it to finish.

Tasks in async_std

Tasks in `async_std` are one of the core abstractions. Much like Rust's `thread`s, they provide some practical functionality over the raw concept. `Tasks` have a relationship to the runtime, but they are in themselves separate. `async_std` tasks have a number of desirable properties:

- They are allocated in one single allocation
- All tasks have a *backchannel*, which allows them to propagate results and errors to the spawning task through the `JoinHandle`
- They carry useful metadata for debugging
- They support task local storage

`async_std`s task API handles setup and teardown of a backing runtime for you and doesn't rely on a runtime being explicitly started.

Blocking

`Task`s are assumed to run *concurrently*, potentially by sharing a thread of execution. This means that operations blocking an *operating system thread*, such as `std::thread::sleep` or io function from Rust's `std` library will *stop execution of all tasks sharing this thread*. Other libraries (such as database drivers) have similar behaviour. Note that *blocking the current thread* is not in and of itself bad behaviour, just something that does not mix well with the concurrent execution model of `async-std`. Essentially, never do this:

```
fn main() {
    task::block_on(async {
        // this is std::fs, which blocks
        std::fs::read_to_string("test_file");
    })
}
```

If you want to mix operation kinds, consider putting such blocking operations on a separate `thread`.

Errors and panics

Tasks report errors through normal patterns: If they are fallible, their `Output` should be of kind `Result<T,E>`.

In case of `panic`, behaviour differs depending on whether there's a reasonable part that addresses the `panic`. If not, the program *aborts*.

In practice, that means that `block_on` propagates panics to the blocking component:

```
fn main() {
    task::block_on(async {
        panic!("test");
    });
}
```

```
thread 'async-task-driver' panicked at 'test', examples/panic.rs:8:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace.
```

While panicking a spawned task will abort:

```

task::spawn(async {
    panic!("test");
});

task::block_on(async {
    task::sleep(Duration::from_millis(10000)).await;
})
```

```
thread 'async-task-driver' panicked at 'test', examples/panic.rs:8:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace.
Aborted (core dumped)
```

That might seem odd at first, but the other option would be to silently ignore panics in spawned tasks. The current behaviour can be changed by catching panics in the spawned task and reacting with custom behaviour. This gives users the choice of panic handling strategy.

Conclusion

`async_std` comes with a useful `Task` type that works with an API similar to `std::thread`. It covers error and panic behaviour in a structured and defined way.

Tasks are separate concurrent units and sometimes they need to communicate. That's where `Stream`s come in.

TODO: Async read/write

TODO: Streams

Tutorial: Writing a chat

Nothing is simpler than creating a chat server, right? Not quite, chat servers expose you to all the fun of asynchronous programming:

How will the server handle clients connecting concurrently?

How will it handle them disconnecting?

How will it distribute the messages?

This tutorial explains how to write a chat server in `async-std`.

You can also find the tutorial in our repository.

Specification and Getting Started

Specification

The chat uses a simple text protocol over TCP. The protocol consists of utf-8 messages, separated by `\n`.

The client connects to the server and sends login as a first line. After that, the client can send messages to other clients using the following syntax:

```
login1, login2, ... loginN: message
```

Each of the specified clients then receives a `from login: message` message.

A possible session might look like this

```
On Alice's computer:   |   On Bob's computer:

> alice                |   > bob
> bob: hello               < from alice: hello
```

```
                        |   > alice, bob: hi!
                            < from bob: hi!
< from bob: hi!         |
```

The main challenge for the chat server is keeping track of many concurrent connections. The main challenge for the chat client is managing concurrent outgoing messages, incoming messages and user's typing.

Getting Started

Let's create a new Cargo project:

```
$ cargo new a-chat
$ cd a-chat
```

Add the following lines to `Cargo.toml`:

```
[dependencies]
futures = "0.3.0"
async-std = "1"
```

Writing an Accept Loop

Let's implement the scaffold of the server: a loop that binds a TCP socket to an address and starts accepting connections.

First of all, let's add required import boilerplate:

```
use async_std::{
    prelude::*, // 1
    task, // 2
    net::{TcpListener, ToSocketAddrs}, // 3
};

type Result<T> = std::result::Result<T, Box<dyn std::error::Error + Send + Syn
```

1. `prelude` re-exports some traits required to work with futures and streams.
2. The `task` module roughly corresponds to the `std::thread` module, but tasks are much lighter weight. A single thread can run many tasks.
3. For the socket type, we use `TcpListener` from `async_std`, which is just like `std::net::TcpListener`, but is non-blocking and uses `async` API.

4. We will skip implementing comprehensive error handling in this example. To propagate the errors, we will use a boxed error trait object. Do you know that there's `From<&'_ str> for Box<dyn Error>` implementation in stdlib, which allows you to use strings with `?` operator?

Now we can write the server's accept loop:

```
async fn accept_loop(addr: impl ToSocketAddrs) -> Result<()> { // 1

    let listener = TcpListener::bind(addr).await?; // 2
    let mut incoming = listener.incoming();
    while let Some(stream) = incoming.next().await { // 3
        // TODO
    }
    Ok(())
}
```

1. We mark the `accept_loop` function as `async`, which allows us to use `.await` syntax inside.
2. `TcpListener::bind` call returns a future, which we `.await` to extract the `Result`, and then `?` to get a `TcpListener`. Note how `.await` and `?` work nicely together. This is exactly how `std::net::TcpListener` works, but with `.await` added. Mirroring API of `std` is an explicit design goal of `async_std`.
3. Here, we would like to iterate incoming sockets, just how one would do in `std`:

```
let listener: std::net::TcpListener = unimplemented!();
for stream in listener.incoming() {
}
```

Unfortunately this doesn't quite work with `async` yet, because there's no support for `async` for-loops in the language yet. For this reason we have to implement the loop manually, by using `while let Some(item) = iter.next().await` pattern.

Finally, let's add main:

```
// main
fn run() -> Result<()> {
    let fut = accept_loop("127.0.0.1:8080");
    task::block_on(fut)
}
```

The crucial thing to realise that is in Rust, unlike other languages, calling an async function does **not** run any code. Async functions only construct futures, which are inert state machines. To start stepping through the future state-machine in an async function, you should use `.await`. In a non-async function, a way to execute a future is to hand it

to the executor. In this case, we use `task::block_on` to execute a future on the current thread and block until it's done.

Receiving messages

Let's implement the receiving part of the protocol. We need to:

1. split incoming `TcpStream` on `\n` and decode bytes as utf-8
2. interpret the first line as a login
3. parse the rest of the lines as a `login: message`

```
use async_std::{
    io::BufReader,
    net::TcpStream,
};

async fn accept_loop(addr: impl ToSocketAddrs) -> Result<()> {
    let listener = TcpListener::bind(addr).await?;
    let mut incoming = listener.incoming();
    while let Some(stream) = incoming.next().await {
        let stream = stream?;
        println!("Accepting from: {}", stream.peer_addr()?);
        let _handle = task::spawn(connection_loop(stream)); // 1
    }
    Ok(())
}

async fn connection_loop(stream: TcpStream) -> Result<()> {
    let reader = BufReader::new(&stream); // 2
    let mut lines = reader.lines();

    let name = match lines.next().await { // 3
        None => Err("peer disconnected immediately")?,
        Some(line) => line?,
    };
    println!("name = {}", name);

    while let Some(line) = lines.next().await { // 4
        let line = line?;
        let (dest, msg) = match line.find(':') { // 5
            None => continue,
            Some(idx) => (&line[..idx], line[idx + 1 ..].trim()),
        };
        let dest: Vec<String> = dest.split(',').map(|name| name.trim().to_stri
        let msg: String = msg.to_string();
    }
    Ok(())
}
```

1. We use `task::spawn` function to spawn an independent task for working with each client. That is, after accepting the client the `accept_loop` immediately starts waiting for the next one. This is the core benefit of event-driven architecture: we serve many clients concurrently, without spending many hardware threads.

2. Luckily, the "split byte stream into lines" functionality is already implemented. `.lines()` call returns a stream of `String`'s.

3. We get the first line -- login

4. And, once again, we implement a manual async for loop.

5. Finally, we parse each line into a list of destination logins and the message itself.

Managing Errors

One serious problem in the above solution is that, while we correctly propagate errors in the `connection_loop`, we just drop the error on the floor afterwards! That is, `task::spawn` does not return an error immediately (it can't, it needs to run the future to completion first), only after it is joined. We can "fix" it by waiting for the task to be joined, like this:

```
let handle = task::spawn(connection_loop(stream));
handle.await?
```

The `.await` waits until the client finishes, and `?` propagates the result.

There are two problems with this solution however! *First*, because we immediately await the client, we can only handle one client at time, and that completely defeats the purpose of async! *Second*, if a client encounters an IO error, the whole server immediately exits. That is, a flaky internet connection of one peer brings down the whole chat room!

A correct way to handle client errors in this case is log them, and continue serving other clients. So let's use a helper function for this:

```
fn spawn_and_log_error<F>(fut: F) -> task::JoinHandle<()>
where
    F: Future<Output = Result<()>> + Send + 'static,
{
    task::spawn(async move {
        if let Err(e) = fut.await {
            eprintln!("{}", e)
        }
```

```
    })
}
```

Sending Messages

Now it's time to implement the other half -- sending messages. A most obvious way to implement sending is to give each `connection_loop` access to the write half of `TcpStream` of each other clients. That way, a client can directly `.write_all` a message to recipients. However, this would be wrong: if Alice sends `bob: foo`, and Charley sends `bob: bar`, Bob might actually receive `fobaor`. Sending a message over a socket might require several syscalls, so two concurrent `.write_all`'s might interfere with each other!

As a rule of thumb, only a single task should write to each `TcpStream`. So let's create a `connection_writer_loop` task which receives messages over a channel and writes them to the socket. This task would be the point of serialization of messages. if Alice and Charley send two messages to Bob at the same time, Bob will see the messages in the same order as they arrive in the channel.

```
use futures::channel::mpsc; // 1
use futures::sink::SinkExt;
use std::sync::Arc;

type Sender<T> = mpsc::UnboundedSender<T>; // 2
type Receiver<T> = mpsc::UnboundedReceiver<T>;

async fn connection_writer_loop(
    mut messages: Receiver<String>,
    stream: Arc<TcpStream>, // 3
) -> Result<()> {
    let mut stream = &*stream;
    while let Some(msg) = messages.next().await {
        stream.write_all(msg.as_bytes()).await?;
    }
    Ok(())
}
```

1. We will use channels from the `futures` crate.
2. For simplicity, we will use `unbounded` channels, and won't be discussing backpressure in this tutorial.
3. As `connection_loop` and `connection_writer_loop` share the same `TcpStream`, we need to put it into an `Arc`. Note that because `client` only reads from the stream and `connection_writer_loop` only writes to the stream, we don't get a race here.

Connecting Readers and Writers

So how do we make sure that messages read in `connection_loop` flow into the relevant `connection_writer_loop`? We should somehow maintain a `peers: HashMap<String, Sender<String>>` map which allows a client to find destination channels. However, this map would be a bit of shared mutable state, so we'll have to wrap an `RwLock` over it and answer tough questions of what should happen if the client joins at the same moment as it receives a message.

One trick to make reasoning about state simpler comes from the actor model. We can create a dedicated broker task which owns the `peers` map and communicates with other tasks using channels. By hiding `peers` inside such an "actor" task, we remove the need for mutexes and also make the serialization point explicit. The order of events "Bob sends message to Alice" and "Alice joins" is determined by the order of the corresponding events in the broker's event queue.

```
use std::collections::hash_map::{Entry, HashMap};

#[derive(Debug)]
enum Event { // 1
    NewPeer {
        name: String,
        stream: Arc<TcpStream>,
    },
    Message {
        from: String,
        to: Vec<String>,
        msg: String,
    },
}

async fn broker_loop(mut events: Receiver<Event>) -> Result<()> {
    let mut peers: HashMap<String, Sender<String>> = HashMap::new(); // 2

    while let Some(event) = events.next().await {
        match event {
            Event::Message { from, to, msg } => {  // 3
                for addr in to {
                    if let Some(peer) = peers.get_mut(&addr) {
                        let msg = format!("from {}: {}\n", from, msg);
                        peer.send(msg).await?
                    }
                }
            }
            Event::NewPeer { name, stream } => {
                match peers.entry(name) {
                    Entry::Occupied(..) => (),
                    Entry::Vacant(entry) => {
```

```
                        let (client_sender, client_receiver) = mpsc::unbounded
                        entry.insert(client_sender); // 4
                        spawn_and_log_error(connection_writer_loop(client_rece
                    }
                }
            }
        }
    }
    Ok(())
}
```

1. The broker task should handle two types of events: a message or an arrival of a new peer.
2. The internal state of the broker is a `HashMap`. Note how we don't need a `Mutex` here and can confidently say, at each iteration of the broker's loop, what is the current set of peers
3. To handle a message, we send it over a channel to each destination
4. To handle a new peer, we first register it in the peer's map ...
5. ... and then spawn a dedicated task to actually write the messages to the socket.

All Together

At this point, we only need to start the broker to get a fully-functioning (in the happy case!) chat:

```
use async_std::{
    io::BufReader,
    net::{TcpListener, TcpStream, ToSocketAddrs},
    prelude::*,
    task,
};
use futures::channel::mpsc;
use futures::sink::SinkExt;
use std::{
    collections::hash_map::{HashMap, Entry},
    sync::Arc,
};

type Result<T> = std::result::Result<T, Box<dyn std::error::Error + Send + Syn
type Sender<T> = mpsc::UnboundedSender<T>;
type Receiver<T> = mpsc::UnboundedReceiver<T>;

// main
fn run() -> Result<()> {
    task::block_on(accept_loop("127.0.0.1:8080"))
}
```

```rust
fn spawn_and_log_error<F>(fut: F) -> task::JoinHandle<()>
where
    F: Future<Output = Result<()>> + Send + 'static,
{
    task::spawn(async move {
        if let Err(e) = fut.await {
            eprintln!("{}", e)
        }
    })
}

async fn accept_loop(addr: impl ToSocketAddrs) -> Result<()> {
    let listener = TcpListener::bind(addr).await?;

    let (broker_sender, broker_receiver) = mpsc::unbounded(); // 1
    let _broker_handle = task::spawn(broker_loop(broker_receiver));
    let mut incoming = listener.incoming();
    while let Some(stream) = incoming.next().await {
        let stream = stream?;
        println!("Accepting from: {}", stream.peer_addr()?);
        spawn_and_log_error(connection_loop(broker_sender.clone(), stream));
    }
    Ok(())
}

async fn connection_loop(mut broker: Sender<Event>, stream: TcpStream) -> Resu
    let stream = Arc::new(stream); // 2
    let reader = BufReader::new(&*stream);
    let mut lines = reader.lines();

    let name = match lines.next().await {
        None => Err("peer disconnected immediately")?,
        Some(line) => line?,
    };
    broker.send(Event::NewPeer { name: name.clone(), stream: Arc::clone(&strea
        .unwrap();

    while let Some(line) = lines.next().await {
        let line = line?;
        let (dest, msg) = match line.find(':') {
            None => continue,
            Some(idx) => (&line[..idx], line[idx + 1 ..].trim()),
        };
        let dest: Vec<String> = dest.split(',').map(|name| name.trim().to_stri
        let msg: String = msg.to_string();

        broker.send(Event::Message { // 4
            from: name.clone(),
            to: dest,
            msg,
        }).await.unwrap();
    }
    Ok(())
}

async fn connection_writer_loop(
```

```
    mut messages: Receiver<String>,
    stream: Arc<TcpStream>,
) -> Result<()> {
    let mut stream = &*stream;
    while let Some(msg) = messages.next().await {
        stream.write_all(msg.as_bytes()).await?;
    }
    Ok(())
}

#[derive(Debug)]
enum Event {
    NewPeer {
        name: String,
        stream: Arc<TcpStream>,
    },
    Message {
        from: String,
        to: Vec<String>,
        msg: String,
    },
}

async fn broker_loop(mut events: Receiver<Event>) -> Result<()> {
    let mut peers: HashMap<String, Sender<String>> = HashMap::new();

    while let Some(event) = events.next().await {
        match event {
            Event::Message { from, to, msg } => {
                for addr in to {
                    if let Some(peer) = peers.get_mut(&addr) {
                        let msg = format!("from {}: {}\n", from, msg);
                        peer.send(msg).await?
                    }
                }
            }
            Event::NewPeer { name, stream} => {
                match peers.entry(name) {
                    Entry::Occupied(..) => (),
                    Entry::Vacant(entry) => {
                        let (client_sender, client_receiver) = mpsc::unbounded
                        entry.insert(client_sender);
                        spawn_and_log_error(connection_writer_loop(client_rece
                    }
                }
            }
        }
    }
    Ok(())
}
```

1. Inside the `accept_loop`, we create the broker's channel and `task`.
2. Inside `connection_loop`, we need to wrap `TcpStream` into an `Arc`, to be able to share it with the `connection_writer_loop`.

3. On login, we notify the broker. Note that we `.unwrap` on send: broker should outlive all the clients and if that's not the case the broker probably panicked, so we can escalate the panic as well.
4. Similarly, we forward parsed messages to the broker, assuming that it is alive.

Clean Shutdown

One of the problems of the current implementation is that it doesn't handle graceful shutdown. If we break from the accept loop for some reason, all in-flight tasks are just dropped on the floor. A more correct shutdown sequence would be:

1. Stop accepting new clients
2. Deliver all pending messages
3. Exit the process

A clean shutdown in a channel based architecture is easy, although it can appear a magic trick at first. In Rust, receiver side of a channel is closed as soon as all senders are dropped. That is, as soon as producers exit and drop their senders, the rest of the system shuts down naturally. In `async_std` this translates to two rules:

1. Make sure that channels form an acyclic graph.
2. Take care to wait, in the correct order, until intermediate layers of the system process pending messages.

In `a-chat`, we already have an unidirectional flow of messages: `reader -> broker -> writer`. However, we never wait for broker and writers, which might cause some messages to get dropped. Let's add waiting to the server:

```
async fn accept_loop(addr: impl ToSocketAddrs) -> Result<()> {
    let listener = TcpListener::bind(addr).await?;

    let (broker_sender, broker_receiver) = mpsc::unbounded();
    let broker_handle = task::spawn(broker_loop(broker_receiver));
    let mut incoming = listener.incoming();
    while let Some(stream) = incoming.next().await {
        let stream = stream?;
        println!("Accepting from: {}", stream.peer_addr()?);
        spawn_and_log_error(connection_loop(broker_sender.clone(), stream));
    }
    drop(broker_sender); // 1
    broker_handle.await?; // 5
    Ok(())
}
```

And to the broker:

```
async fn broker_loop(mut events: Receiver<Event>) -> Result<()> {
    let mut writers = Vec::new();
    let mut peers: HashMap<String, Sender<String>> = HashMap::new();
    while let Some(event) = events.next().await { // 2
        match event {
            Event::Message { from, to, msg } => {
                for addr in to {
                    if let Some(peer) = peers.get_mut(&addr) {
                        let msg = format!("from {}: {}\n", from, msg);
                        peer.send(msg).await?
                    }
                }
            }
            Event::NewPeer { name, stream} => {
                match peers.entry(name) {
                    Entry::Occupied(..) => (),
                    Entry::Vacant(entry) => {
                        let (client_sender, client_receiver) = mpsc::unbounded
                        entry.insert(client_sender);
                        let handle = spawn_and_log_error(connection_writer_loo
                        writers.push(handle); // 4
                    }
                }
            }
        }
    }
    drop(peers); // 3
    for writer in writers { // 4
        writer.await;
    }
    Ok(())
}
```

Notice what happens with all of the channels once we exit the accept loop:

1. First, we drop the main broker's sender. That way when the readers are done, there's no sender for the broker's channel, and the chanel closes.
2. Next, the broker exits `while let Some(event) = events.next().await` loop.
3. It's crucial that, at this stage, we drop the `peers` map. This drops writer's senders.
4. Now we can join all of the writers.
5. Finally, we join the broker, which also guarantees that all the writes have terminated.

Handling Disconnections

Currently, we only ever *add* new peers to the map. This is clearly wrong: if a peer closes connection to the chat, we should not try to send any more messages to it.

One subtlety with handling disconnection is that we can detect it either in the reader's task, or in the writer's task. The most obvious solution here is to just remove the peer from the `peers` map in both cases, but this would be wrong. If *both* read and write fail, we'll remove the peer twice, but it can be the case that the peer reconnected between the two failures! To fix this, we will only remove the peer when the write side finishes. If the read side finishes we will notify the write side that it should stop as well. That is, we need to add an ability to signal shutdown for the writer task.

One way to approach this is a `shutdown: Receiver<()>` channel. There's a more minimal solution however, which makes clever use of RAII. Closing a channel is a synchronization event, so we don't need to send a shutdown message, we can just drop the sender. This way, we statically guarantee that we issue shutdown exactly once, even if we early return via `?` or panic.

First, let's add a shutdown channel to the `connection_loop`:

```
#[derive(Debug)]
enum Void {} // 1

#[derive(Debug)]
enum Event {
    NewPeer {
        name: String,
        stream: Arc<TcpStream>,
        shutdown: Receiver<Void>, // 2
    },
    Message {
        from: String,
        to: Vec<String>,
        msg: String,
    },
}

async fn connection_loop(mut broker: Sender<Event>, stream: Arc<TcpStream>) ->
    // ...
    let (_shutdown_sender, shutdown_receiver) = mpsc::unbounded::<Void>(); //
    broker.send(Event::NewPeer {
        name: name.clone(),
        stream: Arc::clone(&stream),
        shutdown: shutdown_receiver,
    }).await.unwrap();
    // ...
}
```

1. To enforce that no messages are sent along the shutdown channel, we use an uninhabited type.
2. We pass the shutdown channel to the writer task.
3. In the reader, we create a `_shutdown_sender` whose only purpose is to get

dropped.

In the `connection_writer_loop`, we now need to choose between shutdown and message channels. We use the `select` macro for this purpose:

```
use futures::{select, FutureExt};

async fn connection_writer_loop(
    messages: &mut Receiver<String>,
    stream: Arc<TcpStream>,
    shutdown: Receiver<Void>, // 1
) -> Result<()> {
    let mut stream = &*stream;
    let mut messages = messages.fuse();
    let mut shutdown = shutdown.fuse();
    loop { // 2
        select! {
            msg = messages.next().fuse() => match msg { // 3
                Some(msg) => stream.write_all(msg.as_bytes()).await?,
                None => break,
            },
            void = shutdown.next().fuse() => match void {
                Some(void) => match void {}, // 4
                None => break,
            }
        }
    }
    Ok(())
}
```

1. We add shutdown channel as an argument.
2. Because of `select`, we can't use a `while let` loop, so we desugar it further into a `loop`.
3. Function fuse() is used to turn any `Stream` into a `FusedStream`. This is used for fusing a stream such that poll_next will never again be called once it has finished.
4. In the shutdown case we use `match void {}` as a statically-checked `unreachable!()`.

Another problem is that between the moment we detect disconnection in `connection_writer_loop` and the moment when we actually remove the peer from the `peers` map, new messages might be pushed into the peer's channel. To not lose these messages completely, we'll return the messages channel back to the broker. This also allows us to establish a useful invariant that the message channel strictly outlives the peer in the `peers` map, and makes the broker itself infallible.

Final Code

The final code looks like this:

```
use async_std::{
    io::BufReader,
    net::{TcpListener, TcpStream, ToSocketAddrs},
    prelude::*,
    task,
};
use futures::channel::mpsc;
use futures::sink::SinkExt;
use futures::{select, FutureExt};
use std::{
    collections::hash_map::{Entry, HashMap},
    future::Future,
    sync::Arc,
};

type Result<T> = std::result::Result<T, Box<dyn std::error::Error + Send + Syn
type Sender<T> = mpsc::UnboundedSender<T>;
type Receiver<T> = mpsc::UnboundedReceiver<T>;

#[derive(Debug)]
enum Void {}

// main
fn run() -> Result<()> {
    task::block_on(accept_loop("127.0.0.1:8080"))
}

async fn accept_loop(addr: impl ToSocketAddrs) -> Result<()> {
    let listener = TcpListener::bind(addr).await?;
    let (broker_sender, broker_receiver) = mpsc::unbounded();
    let broker_handle = task::spawn(broker_loop(broker_receiver));
    let mut incoming = listener.incoming();
    while let Some(stream) = incoming.next().await {
        let stream = stream?;
        println!("Accepting from: {}", stream.peer_addr()?);
        spawn_and_log_error(connection_loop(broker_sender.clone(), stream));
    }
    drop(broker_sender);
    broker_handle.await;
    Ok(())
}

async fn connection_loop(mut broker: Sender<Event>, stream: TcpStream) -> Resu
    let stream = Arc::new(stream);
    let reader = BufReader::new(&*stream);
    let mut lines = reader.lines();

    let name = match lines.next().await {
        None => Err("peer disconnected immediately")?,
        Some(line) => line?,
    };
    let (_shutdown_sender, shutdown_receiver) = mpsc::unbounded::<Void>();
```

```
    broker.send(Event::NewPeer {
        name: name.clone(),
        stream: Arc::clone(&stream),
        shutdown: shutdown_receiver,
    }).await.unwrap();

    while let Some(line) = lines.next().await {
        let line = line?;
        let (dest, msg) = match line.find(':') {
            None => continue,
            Some(idx) => (&line[..idx], line[idx + 1 ..].trim()),
        };
        let dest: Vec<String> = dest.split(',').map(|name| name.trim().to_stri
        let msg: String = msg.trim().to_string();

        broker.send(Event::Message {
            from: name.clone(),
            to: dest,
            msg,
        }).await.unwrap();
    }

    Ok(())
}

async fn connection_writer_loop(
    messages: &mut Receiver<String>,
    stream: Arc<TcpStream>,
    shutdown: Receiver<Void>,
) -> Result<()> {
    let mut stream = &*stream;
    let mut messages = messages.fuse();
    let mut shutdown = shutdown.fuse();
    loop {
        select! {
            msg = messages.next().fuse() => match msg {
                Some(msg) => stream.write_all(msg.as_bytes()).await?,
                None => break,
            },
            void = shutdown.next().fuse() => match void {
                Some(void) => match void {},
                None => break,
            }
        }
    }
    Ok(())
}

#[derive(Debug)]
enum Event {
    NewPeer {
        name: String,
        stream: Arc<TcpStream>,
        shutdown: Receiver<Void>,
    },
    Message {
```

```
        from: String,
        to: Vec<String>,
        msg: String,
    },
}

async fn broker_loop(events: Receiver<Event>) {
    let (disconnect_sender, mut disconnect_receiver) = // 1
        mpsc::unbounded::<(String, Receiver<String>)>();
    let mut peers: HashMap<String, Sender<String>> = HashMap::new();
    let mut events = events.fuse();
    loop {
        let event = select! {
            event = events.next().fuse() => match event {
                None => break, // 2
                Some(event) => event,
            },
            disconnect = disconnect_receiver.next().fuse() => {
                let (name, _pending_messages) = disconnect.unwrap(); // 3
                assert!(peers.remove(&name).is_some());
                continue;
            },
        };
        match event {
            Event::Message { from, to, msg } => {
                for addr in to {
                    if let Some(peer) = peers.get_mut(&addr) {
                        let msg = format!("from {}: {}\n", from, msg);
                        peer.send(msg).await
                            .unwrap() // 6
                    }
                }
            }
            Event::NewPeer { name, stream, shutdown } => {
                match peers.entry(name.clone()) {
                    Entry::Occupied(..) => (),
                    Entry::Vacant(entry) => {
                        let (client_sender, mut client_receiver) = mpsc::unbou
                        entry.insert(client_sender);
                        let mut disconnect_sender = disconnect_sender.clone();
                        spawn_and_log_error(async move {
                            let res = connection_writer_loop(&mut client_recei
                            disconnect_sender.send((name, client_receiver)).aw
                                .unwrap();
                            res
                        });
                    }
                }
            }
        }
    }
    drop(peers); // 5
    drop(disconnect_sender); // 6
    while let Some((_name, _pending_messages)) = disconnect_receiver.next().aw
    }
}
```

```
fn spawn_and_log_error<F>(fut: F) -> task::JoinHandle<()>
where
    F: Future<Output = Result<()>> + Send + 'static,
{
    task::spawn(async move {
        if let Err(e) = fut.await {
            eprintln!("{}", e)
        }
    })
}
```

1. In the broker, we create a channel to reap disconnected peers and their undelivered messages.
2. The broker's main loop exits when the input events channel is exhausted (that is, when all readers exit).
3. Because broker itself holds a `disconnect_sender`, we know that the disconnections channel can't be fully drained in the main loop.
4. We send peer's name and pending messages to the disconnections channel in both the happy and the not-so-happy path. Again, we can safely unwrap because the broker outlives writers.
5. We drop `peers` map to close writers' messages channel and shut down the writers for sure. It is not strictly necessary in the current setup, where the broker waits for readers' shutdown anyway. However, if we add a server-initiated shutdown (for example, kbd:[ctrl+c] handling), this will be a way for the broker to shutdown the writers.
6. Finally, we close and drain the disconnections channel.

Implementing a client

Since the protocol is line-based, implementing a client for the chat is straightforward:

- Lines read from stdin should be sent over the socket.
- Lines read from the socket should be echoed to stdout.

Although async does not significantly affect client performance (as unlike the server, the client interacts solely with one user and only needs limited concurrency), async is still useful for managing concurrency!

The client has to read from stdin and the socket *simultaneously*. Programming this with threads is cumbersome, especially when implementing a clean shutdown. With async, the `select!` macro is all that is needed.

```
use async_std::{
```

```
    io::{stdin, BufReader},
    net::{TcpStream, ToSocketAddrs},
    prelude::*,
    task,
};
use futures::{select, FutureExt};

type Result<T> = std::result::Result<T, Box<dyn std::error::Error + Send + Syn

// main
fn run() -> Result<()> {
    task::block_on(try_run("127.0.0.1:8080"))
}

async fn try_run(addr: impl ToSocketAddrs) -> Result<()> {
    let stream = TcpStream::connect(addr).await?;
    let (reader, mut writer) = (&stream, &stream); // 1
    let mut lines_from_server = BufReader::new(reader).lines().fuse(); // 2
    let mut lines_from_stdin = BufReader::new(stdin()).lines().fuse(); // 2
    loop {
        select! { // 3
            line = lines_from_server.next().fuse() => match line {
                Some(line) => {
                    let line = line?;
                    println!("{}", line);
                },
                None => break,
            },
            line = lines_from_stdin.next().fuse() => match line {
                Some(line) => {
                    let line = line?;
                    writer.write_all(line.as_bytes()).await?;
                    writer.write_all(b"\n").await?;
                }
                None => break,
            }
        }
    }
    Ok(())
}
```

1. Here we split `TcpStream` into read and write halves: there's `impl AsyncRead for &'_ TcpStream`, just like the one in std.
2. We create a stream of lines for both the socket and stdin.
3. In the main select loop, we print the lines we receive from the server and send the lines we read from the console.

Patterns

This section documents small, useful patterns.

It is intended to be read at a glance, allowing you to get back when you have a problem.

Small Patterns

A collection of small, useful patterns.

Splitting streams

`async-std` doesn't provide a `split()` method on `io` handles. Instead, splitting a stream into a read and write half can be done like this:

```
use async_std::{io, net::TcpStream};
async fn echo(stream: TcpStream) {
    let (reader, writer) = &mut (&stream, &stream);
    io::copy(reader, writer).await;
}
```

Production-Ready Accept Loop

A production-ready accept loop needs the following things:

1. Handling errors
2. Limiting the number of simultaneous connections to avoid deny-of-service (DoS) attacks

Handling errors

There are two kinds of errors in an accept loop:

1. Per-connection errors. The system uses them to notify that there was a connection in the queue and it's dropped by the peer. Subsequent connections can be already queued so next connection must be accepted immediately.
2. Resource shortages. When these are encountered it doesn't make sense to accept the next socket immediately. But the listener stays active, so you server should try to accept socket later.

Here is the example of a per-connection error (printed in normal and debug mode):

```
Error: Connection reset by peer (os error 104)
Error: Os { code: 104, kind: ConnectionReset, message: "Connection reset by pe
```

And the following is the most common example of a resource shortage error:

```
Error: Too many open files (os error 24)
Error: Os { code: 24, kind: Other, message: "Too many open files" }
```

Testing Application

To test your application for these errors try the following (this works on unixes only).

Lower limits and start the application:

```
$ ulimit -n 100
$ cargo run --example your_app
   Compiling your_app v0.1.0 (/work)
    Finished dev [unoptimized + debuginfo] target(s) in 5.47s
     Running `target/debug/examples/your_app`
Server is listening on: http://127.0.0.1:1234
```

Then in another console run the `wrk` benchmark tool:

```
$ wrk -c 1000 http://127.0.0.1:1234
Running 10s test @ http://localhost:8080/
  2 threads and 1000 connections
$ telnet localhost 1234
Trying ::1...
Connected to localhost.
```

Important is to check the following things:

1. The application doesn't crash on error (but may log errors, see below)
2. It's possible to connect to the application again once load is stopped (few seconds after `wrk`). This is what `telnet` does in example above, make sure it prints `Connected to <hostname>`.
3. The `Too many open files` error is logged in the appropriate log. This requires to set "maximum number of simultaneous connections" parameter (see below) of your application to a value greater then `100` for this example.
4. Check CPU usage of the app while doing a test. It should not occupy 100% of a single CPU core (it's unlikely that you can exhaust CPU by 1000 connections in Rust, so this means error handling is not right).

Testing non-HTTP applications

If it's possible, use the appropriate benchmark tool and set the appropriate number of connections. For example `redis-benchmark` has a `-c` parameter for that, if you

implement redis protocol.

Alternatively, can still use `wrk`, just make sure that connection is not immediately closed. If it is, put a temporary timeout before handing the connection to the protocol handler, like this:

```
while let Some(stream) = incoming.next().await {
    task::spawn(async {
        task::sleep(Duration::from_secs(10)).await; // 1
        connection_loop(stream).await;
    });
}
```

1. Make sure the sleep coroutine is inside the spawned task, not in the loop.

Handling Errors Manually

Here is how basic accept loop could look like:

```
async fn accept_loop(addr: impl ToSocketAddrs) -> Result<()> {
    let listener = TcpListener::bind(addr).await?;
    let mut incoming = listener.incoming();
    while let Some(result) = incoming.next().await {
        let stream = match result {
            Err(ref e) if is_connection_error(e) => continue, // 1
            Err(e) => {
                eprintln!("Error: {}. Pausing for 500ms.", e); // 3
                task::sleep(Duration::from_millis(500)).await; // 2
                continue;
            }
            Ok(s) => s,
        };
        // body
    }
    Ok(())
}
```

1. Ignore per-connection errors.
2. Sleep and continue on resource shortage.
3. It's important to log the message, because these errors commonly mean the misconfiguration of the system and are helpful for operations people running the application.

Be sure to test your application.

External Crates

The crate `async-listen` has a helper to achieve this task:

```
use async_listen::{ListenExt, error_hint};

async fn accept_loop(addr: impl ToSocketAddrs) -> Result<()> {

    let listener = TcpListener::bind(addr).await?;
    let mut incoming = listener
        .incoming()
        .log_warnings(log_accept_error) // 1
        .handle_errors(Duration::from_millis(500));
    while let Some(socket) = incoming.next().await { // 2
        // body
    }
    Ok(())
}

fn log_accept_error(e: &io::Error) {
    eprintln!("Error: {}. Listener paused for 0.5s. {}", e, error_hint(e)) //
}
```

1. Logs resource shortages (`async-listen` calls them warnings). If you use `log` crate or any other in your app this should go to the log.
2. Stream yields sockets without `Result` wrapper after `handle_errors` because all errors are already handled.
3. Together with the error we print a hint, which explains some errors for end users. For example, it recommends increasing open file limit and gives a link.

Be sure to test your application.

Connections Limit

Even if you've applied everything described in Handling Errors section, there is still a problem.

Let's imagine you have a server that needs to open a file to process client request. At some point, you might encounter the following situation:

1. There are as many client connection as max file descriptors allowed for the application.
2. Listener gets `Too many open files` error so it sleeps.
3. Some client sends a request via the previously open connection.
4. Opening a file to serve request fails, because of the same `Too many open files` error, until some other client drops a connection.

There are many more possible situations, this is just a small illustation that limiting number of connections is very useful. Generally, it's one of the ways to control resources used by a server and avoiding some kinds of deny of service (DoS) attacks.

`async-listen` crate

Limiting maximum number of simultaneous connections with `async-listen` looks like the following:

```
use async_listen::{ListenExt, Token, error_hint};

async fn accept_loop(addr: impl ToSocketAddrs) -> Result<()> {

    let listener = TcpListener::bind(addr).await?;
    let mut incoming = listener
        .incoming()
        .log_warnings(log_accept_error)
        .handle_errors(Duration::from_millis(500)) // 1
        .backpressure(100);
    while let Some((token, socket)) = incoming.next().await { // 2
         task::spawn(async move {
             connection_loop(&token, stream).await; // 3
         });
    }
    Ok(())
}
async fn connection_loop(_token: &Token, stream: TcpStream) { // 4
    // ...
}
```

1. We need to handle errors first, because `backpressure` helper expects stream of `TcpStream` rather than `Result`.
2. The token yielded by a new stream is what is counted by backpressure helper. I.e. if you drop a token, new connection can be established.
3. We give the connection loop a reference to token to bind token's lifetime to the lifetime of the connection.
4. The token itsellf in the function can be ignored, hence `_token`

Be sure to test this behavior.

Security

Writing a highly perfomant async core library is a task involving some instances of unsafe code.

We take great care in vetting all unsafe code included in `async-std` and do follow

generally accepted practices.

In the case that you find a security-related bug in our library, please get in touch with our security contact.

Patches improving the resilience of the library or the testing setup are happily accepted on our github org.

Policy

Safety is one of the core principles of what we do, and to that end, we would like to ensure that async-std has a secure implementation. Thank you for taking the time to responsibly disclose any issues you find.

All security bugs in async-std distribution should be reported by email to florian.gilcher@ferrous-systems.com. This list is delivered to a small security team. Your email will be acknowledged within 24 hours, and you'll receive a more detailed response to your email within 48 hours indicating the next steps in handling your report. If you would like, you can encrypt your report using our public key. This key is also On MIT's keyserver and reproduced below.

Be sure to use a descriptive subject line to avoid having your report be missed. After the initial reply to your report, the security team will endeavor to keep you informed of the progress being made towards a fix and full announcement. As recommended by RFPolicy, these updates will be sent at least every five days. In reality, this is more likely to be every 24-48 hours.

If you have not received a reply to your email within 48 hours, or have not heard from the security team for the past five days, there are a few steps you can take (in order):

- Post on our Community forums

Please note that the discussion forums are public areas. When escalating in these venues, please do not discuss your issue. Simply say that you're trying to get a hold of someone from the security team.

Disclosure policy

The async-std project has a 5 step disclosure process.

- The security report is received and is assigned a primary handler. This person will coordinate the fix and release process.
- The problem is confirmed and a list of all affected versions is determined.

- Code is audited to find any potential similar problems.
- Fixes are prepared for all releases which are still under maintenance. These fixes are not committed to the public repository but rather held locally pending the announcement.
- On the embargo date, the changes are pushed to the public repository and new builds are deployed to crates.io. Within 6 hours, a copy of the advisory will be published on the the async.rs blog.

This process can take some time, especially when coordination is required with maintainers of other projects. Every effort will be made to handle the bug in as timely a manner as possible, however it's important that we follow the release process above to ensure that the disclosure is handled in a consistent manner.

Credits

This policy is adapted from the Rust project security policy.

PGP Key


```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQENBF1Wu/ABCADJaGt4HwSlqKB9BGHWYKZj/6mTMbmc29vsEOcCSQKo6myCf9zc
sasWAttep4FAUDX+MJhVbBTSq9M1YVxp33Qh5AF0t9SnJZnbI+BZuGawcHDL01xE
bE+8bcA2+szeTTUZCeWwsaoTd/2qmQKvpUCBQp7uBs/ITO/I2q7+xCGXaOHZwUKc
H8SUBLd35nYFtjXAeejoZVkqG2qEjrc9bkZAwxFXi7Fw94QdkNLaCjNfKxZON/qP
A3WOpyWPr3ERk5C5prjEAvrW8kdqpTRjdmzQjsr8UEXb5GGEOo93N4OLZVQ2mXt9
dfn++GOnOk7sTxvfiDH8Ru5o4zCtKgO+r5/LABEBAAG0UkZsb3JpYW4gR2lsY2hl
ciAoU2VjdXJpdHkgY29udGFjdCBhc3luYy1zdGQpIDxmbG9yaWFuLmdpbGNoZXJA
ZmVycm91cy1zeXN0ZW1zLmNvbT6JATgEEwECACIFAl1Wu/ACGwMGCwkIBwMCBhUI
AgkKCwQWAgMBAh4BAheAAAoJEACXY97PwLtSc0AH/18yvrElVOkG0ADWX7l+JKHH
nMQtYj0Auop8d6TuKBbpwtYhwELrQoITDMV7f2XEnchNsvYxAyBZhIISmXeJboE1
KzZD1O+4QPXRcXhj+QNsKQ680mrgZXgAI2Y4ptIW9Vyw3jiHu/ZVopvDAt4li+up
3fRJGPAvGu+tclpJmA+Xam23cDj89M7/wHHgKIyT59WgFwyCgibL+NHKwg2Unzou
9uyZQnq6hf62sQTWEZIAr9BQpKmluplNIJHDeECWzZoE9ucE2ZXsq5pq9qojsAMK
yRdaFdpBcD/AxtrTKFeXGS7X7LqaljY/IFBEdJOqVNWpqSLjGWqjSLIEsc1AB0K5
AQ0EXVa78AEIAJMxBOEEW+2c3CcjFuUfcRsoBsFH3Vk+GwCbjIpNHq/eAvS1yy2L
u10U5CcT5Xb6be3AeCYv00ZHVbEi6VwoauVCSX8qDjhVzQxvNLgQ1SduobjyF6t8
3M/wTija6NvMKszyw1l2oHepxSMLej1m49DyCDFNiZm5rjQcYnFT4J71syxViqHF
v2fWCheTrHP3wfBAt5zyDet7IZd/EhYAK6xXEwr9nBPjfbaVexm2B8K6hOPNj0Bp
OKm4rcOj7JYlcxrwhMvNnwEue7MqH1oXAsoaC1BW+qs4acp/hHpesweL6Rcg1pED
OJUQd3UvRsqRK0EsorDu0oj5wt6Qp3ZEbPMAEQEAAYkBHwQYAQIACQUCXVa78AIb
DAAKCRAAl2Pez8C7Uv8bB/9scRm2wvzHLbFtcEHaHvlKO1yYfSVqKqJzIKHc7pM2
+szM8JVRTxAbzK5Xih9SB5xlekixxO2UCJI5DkJ/ir/RCcg+/CAQ8iLm2UcYAgJD
TocKiR5gjNAvUDI4tMrDLLdF+7+RCQGc7HBSxFiNBJVGAztGVh1+cQ0zaCX6Tt33
1EQtyRcPID0m6+ip5tCJN0dILC0YcwzXGrSgjB03JqItIyJEucdQz6UB84TIAGku
JJl4tktgD9T7Rb5uzRhHCSbLy89DQVvCcKD4B94ffuDW3HO8n8utDusOiZuG4BUf
```

```
WdFy6/gTLNiFbTzkq1BBJQMN1nBwGs1sn63RRgjumZ1N
=dIcF
-----END PGP PUBLIC KEY BLOCK-----
```

Glossary

blocking

"blocked" generally refers to conditions that keep a task from doing its work. For example, it might need data to be sent by a client before continuing. When tasks become blocked, usually, other tasks are scheduled.

Sometimes you hear that you should never call "blocking functions" in an async context. What this refers to is functions that block the current thread and do not yield control back. This keeps the executor from using this thread to schedule another task.